МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС
"ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ"
НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

Тестування: Основні визначення, аксіоми та принципи. Текст лекцій. Частина І

Текст лекцій до курсу «Технології розробки і тестування програм»

для студентів напрямів підготовки:
6.040302 «Інформатика»
6.050101 «Комп’ютерні науки».

Дідковська М.В., Тимошенко Ю.О.

ВСТУП

Як відомо, основним завданням перших десятиріч комп'ютерної ери був розвиток апаратних засобів обчислювальної техніки. Це було обумовлено високою вартістю завдань з обробки та збереження інформації. Наразі, досягнення мікроелектроніки призвели до значного збільшення продуктивності комп'ютерів з одночасним суттєвим зниженням їхньої вартості.

Основним завданням останнього десятиріччя XX століття та початку XXI ст. стало вдосконалення якості комп'ютерних сервісів, можливості яких повністю залежать від програмного забезпечення (ПЗ). Програмне забезпечення розробляють вже понад п'ятдесяти років і за цей період коло задач, які воно може вирішувати, рівень їх складності та форми представлення отриманих результатів кардинально змінилися [1].

На сьогодні розробку ПЗ розглядають вже під кутом зору технології "S+S" (Software plus Services, ініціатива корпорації Microsoft, липень 2007р.), яка передбачає збірку ПЗ разом з сервісами в єдиний, персоналізований, доступний з будь-якого місця інструмент. У той же час й дотепер розробка якісних програмних продуктів не стала нормою, а загальних технологій, з допомогою яких розробники можуть створювати надійне ПЗ з відповідними витратами до заданого часу, — не існує. Джерела несправностей сучасного ПЗ вкрай різноманітні і це лише ускладнює проблему. Одночасно зріс і масштаб цієї проблеми. Якщо у минулому ціна помилки неякісного ПЗ обмежувалась одним користувачем або невеликою групою, то зараз ці рамки суттєво розширилися. Проблеми, що колись зводились до повторного „проходження", наприклад, по звіту у ручному режимі, зараз можуть означати життя чи смерть цілої організації.

Тому актуальність розробки якісного ПЗ підтверджується насамперед економічними чинниками. Як відомо, багато „галузевих стандартів на „добре" комерційне ПЗ передбачають наявність виникнення близько 6 помилок на 1000 рядків коду при середньому показникові у 30 таких помилок". Можна стверджувати, що рівень помилок за останні 20 років практично не змінився, не дивлячись на застосування об'єктно-орієнтованої технології, автоматичних налагоджувачів, більш якісних засобів тестування та більш суворий контроль типів у таких сучасних мовах програмування, як Java, Ada та ін. Згідно з даними звіту Національного інституту по стандартах та технології США [2] „об'єм економічних витрат внаслідок несправного ПЗ у США досягає мільярдів доларів на рік, що складає за деякими оцінками близько 1% національного валового внутрішнього продукту".

За словами Б.Гейтса, актуальність надійних обчислень має, також найвищий пріоритет у розвитку сучасних інформаційних технологій. Ця тенденція отримала спеціальну назву — так звана „платформа для достовірних обчислень" [3]. Тому поняття „надійне ПЗ" означає спроможність ПЗ виконувати покладені на нього функції при надходженні вимог на їх виконання. Відомий розробник складних програмних проектів Дж.Фокс (фірма IBM) навіть стверджував: „Вірне ПЗ не зазнає несправностей" [4].

За думкою іншої видатної постаті у комп'ютерному світі Д.Паттерсона, який втілює у життя створення „відновлюваних (після відмови) комп'ютерних платформ" („Recovery Oriented Computing"), світова гонка лише за продуктивністю електронно-обчислювальних машин призвела лише до залежності людини від технологій. Поведінка комп'ютерів, починаючи з найпростіших пристроїв і до потужних маршрутизаторів, які підтримують інфраструктуру Internet, є непередбачуваною. „Комп'ютери на часі стають всепроникаючими

(„ubiquitous computing”), сучасний світ все більш залежить від них, але ще ніхто не довів, що вони заслуговують на таку довіру”. У своєму всесвітньо відомому маніфесті Д.Паттерсон стверджує: „Настав час обрати принципово іншу основу, на якій будуть побудовані технології майбутнього”. І далі: „Ми повинні створювати інформаційні технології, на які світ дійсно може покластися так, як він спирається на технології інших типів, повністю довіряючи їм”.

Цим виданням автори розпочинають цикл публікацій науково-методичних робіт, присвячених систематичному висвітленню принципів, моделей та методів досягнення якості ПЗ. Сподіваємось, що самостійна робота студентів над цим матеріалом дозволять їм сформувати своє уявлення щодо якості ПЗ, як інженерної дисципліни.

В основу збірки покладено досвід викладання авторами відповідних дисциплін у ННК „ІПСА” та факультеті електроніки НТУУ „КПІ”.

Автори ставили перед собою за мету:

- викласти класичні основи, що віддзеркалюють наявний світовий досвід програмної інженерії при створенні якісного ПЗ;
- показати останні наукові та практичні результати у цій царині;
- забезпечити комплексний розгляд найбільш важливих питань, що виникають при забезпеченні якості у великих програмних системах.

Приступаючи до реалізації цього завдання необхідно прийняти до уваги наступні фактори:

1. За останнє десятиліття досягнуто великого прогресу у розвитку обчислювальної інфраструктури. Нові операційні платформи перевершили старі операційні системи за показником функціональності. З’явилися нові мови програмування, нові API для

комунікацій, захисту, розподілених обчислень і, нарешті, Web. Все це повністю, змінило середовище розробників ПЗ.

2. Хаотичний період розвитку ПЗ, коли значно більше уваги приділялось саме програмному коду, а не його якості, став відходити у минуле. Вже на кінець 80-х років ці зміни стали цілком очевидними. Зокрема, було опубліковано одну з перших робіт, присвячену цьому напрямку — першу книгу з тестування ПЗ [5].

3. Організація виробництва програмних систем та управління цим процесом є окремою гілкою при розробці ПЗ. На відомій конференції підкомітету НАТО з науки і техніки (1968 р.), присвяченій проблемам створення ПЗ для великих програмних проектів, напрямок з управління програмними проектами та питання якості ПЗ, був сформульований вперше. Процеси розробки якісного ПЗ у подальшому викликали необхідність свого постійного удосконалення в зв'язку з помітним посиленням впливу якості ПЗ на життя людей. Як результат, за останні роки програмна індустрія досягла такого рівня розвитку, при якому вимоги до забезпечення якості стали обов'язковим пунктом договорів на предмет розробки програмних систем.

4. Починаючи з 90-х років минулого сторіччя, здійснюються перші реальні спроби щодо перетворення розробки ПЗ в інженерну дисципліну завдяки концепціям CBSE (Component-Based Software Engineering — компонентна розробка ПЗ) та COTS (Commercial Off-The-Shelf — готові комерційно доступні компоненти).

5. Нарешті, задачі створення сучасних програмних систем мають враховувати широко уживані в практиці уніфіковані технології проектування типу RUP (Rational Unified Process) та UML-діаграми.

6. Гарантування якості ПЗ – завдання, рішення якого потребує комплексного дослідження в наступних напрямках:

- розробка засобів аналізу і оцінка якості ПЗ на етапах його життєвого циклу (ЖЦ);
- визначення і управління параметрами, які впливають на якість ПЗ на всіх етапах його ЖЦ.

7. Якість ПЗ визначається декількома параметрами, серед яких надійність ПЗ є основною формалізованою характеристикою. Тому виникає необхідність в забезпеченні надійності ПЗ, починаючи з самих ранніх фаз його ЖЦ [5-8] та з урахуванням сучасних тенденцій в його розробці та проектуванні [9].

Комп'ютерні науки взагалі та програмна інженерія, зокрема – вельми популярні та стрімко прогресуючі царині знань. Тому цілком зрозуміла та прискіплива увага, яку приділяє освіті в галузі комп'ютерних наук світове освітянське товариство. Як результат вже багато десятиліть існують Міжнародні стандарти з комп'ютерної освіти, так звані Computing Curricula. Нині діючий стандарт – це Computing Curricula 2001, який уніфікує та упорядковує знання, вміння та навички, що є необхідними для спеціалістів у цьому напрямку. Міжнародним науковим та педагогічним товариством розроблені також перелік основних знань в сфері програмної інженерії – IEEE/ACM Software Engineering Body of Knowledge (SWEBOK) 2001.

Зміст цих посібників відповідають вимогам цих стандартів.

Запропонований текст лекцій є першою частиною циклу та носить назву: «Технології розробки і тестування програм. Частина 1: Основні визначення, аксіоми та принципи». Видання складається з чотирьох розділів.

У Вступі обґрунтовано актуальність проблеми тестування програм у її історичному розрізі та показано розвиток відповідної дисципліни.

Перший розділ присвячено питанню тестування програмного забезпечення як засобу підвищення надійності функціонування. Представлені основні визначення, розглянуті та проаналізовані рівні та види тестування. На завершення розділу наведені існуючі техніки тестування.

Другий розділ знайомить читача з аксіомами тестування програмного забезпечення.

У третьому розділі досліджується місце тестування у циклі розробки програмного забезпечення. Показані основні джерела виникнення помилок та принципи формування вартості їх виправлення.

Заключний, четвертий, розділ тексту лекцій містить порівняльний аналіз принципів тестування, що сформувалися за останні десятиріччя.

Графічно схема структури тексту лекцій представлена у вигляді мапи знань на рис. 1.

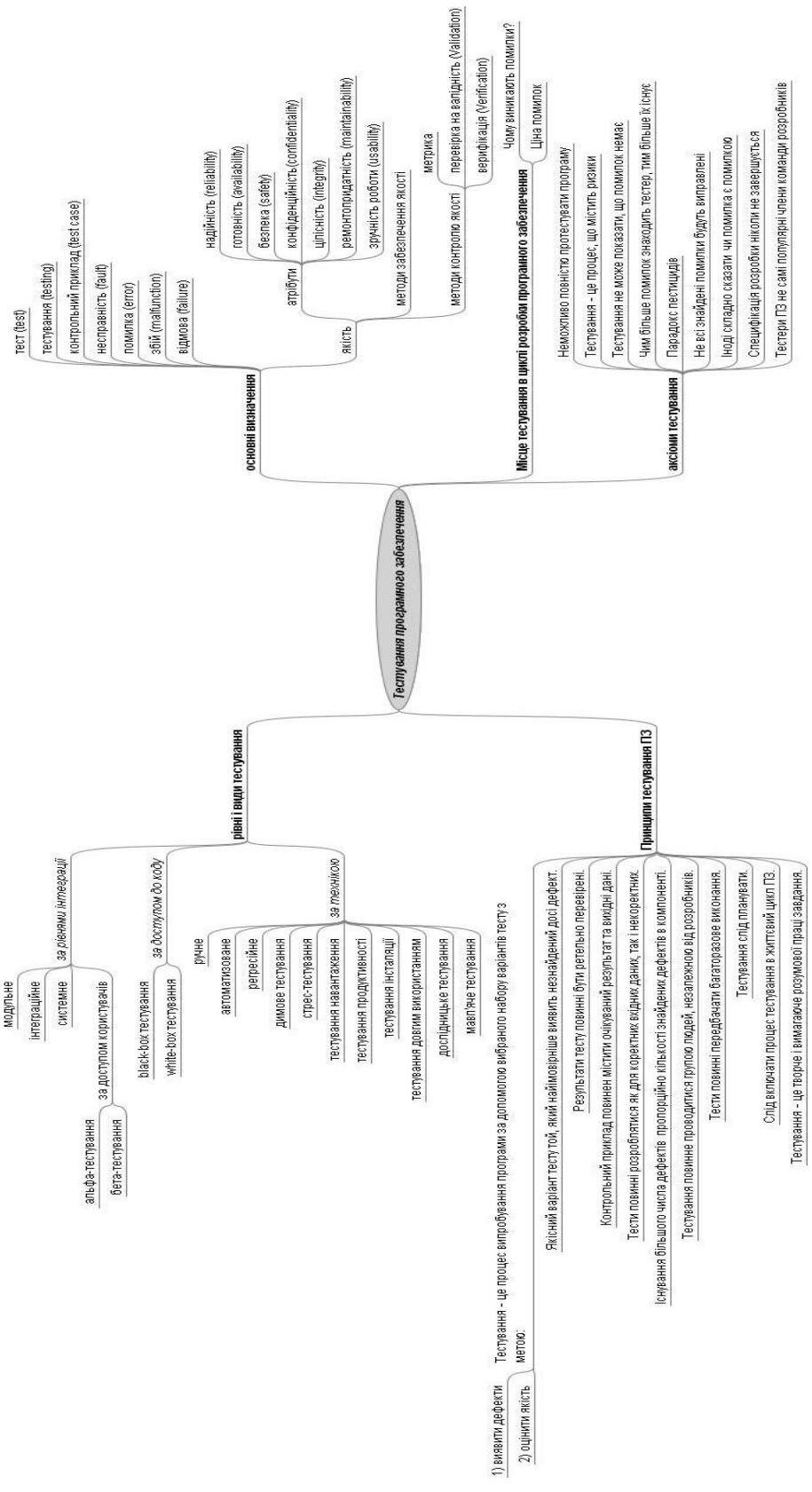

Рис. 1. Структура тексту лекцій.

1. ТЕСТУВАННЯ, ЯК ЗАСІБ ПІДВИЩЕННЯ НАДІЙНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування – це процес керованого експериментування з продуктом за допомогою тестів з метою виявлення в ньому помилок, тобто виявлення неточностей допущених розробниками ПЗ.

Тест – контрольна задача для перевірки коректності функціонування системи та/або її ПЗ. Основна ідея тестування – запустити ПЗ і спостерігати за його роботою та її наслідками. Якщо збій в роботі ПЗ відбувся, тоді аналізується звіт з метою виявлення місцезнаходження помилки, яка його викликала.

“Вдалим” тестом є той, при якому виконання програми закінчилось з помилкою і навпаки. Тестування виконує дві основні задачі[10]:

1) демонстрація якості функціонування ПЗ;
2) знаходження і усунення помилок в ПЗ.

Головною метою тестування є збільшення ймовірності того, що ПЗ, яке тестується, буде відповідати своїм специфікаціям.

Тестування - процес ітераційний. Після виявлення і виправлення кожної помилки обов'язково слідує повторення тестів, що має на меті перевірку працездатності програми. Більш того, для ідентифікації причини виявленої проблеми може бути потрібно проведення спеціального додаткового тестування. При цьому завжди потрібно пам'ятати фундаментальний висновок, зроблений професором Едсжером Дейкстрой в 1972 г: "Тестування програм може служити доказом наявності помилок, але ніколи не доведе їх відсутність!" [11].

Тестування може виконуватися вручну або автоматично. Автоматичне тестування, якщо воно виконано коректно, забезпечує більш швидке, якісне і ефективне тестування, що приводить до зменшення кількості тестів, скороченню часу тестування, підвищення стійкості системи.

Тестування пронизує весь життєвий цикл ПЗ, починаючи від проектування і закінчуючи невизначено довгим етапом експлуатації, та безпосередньо пов'язане з управлінням вимогами і змінами, адже метою тестування якраз є можливість переконатися у відповідності програм заявленим вимогам [10].

Як було зазначено вище, тестування підтверджує, що ПЗ працює згідно специфікації. Вважають, що ***програма працює коректно***, якщо вона задовольняє наступним критеріям:

1) отримавши коректні дані, програма надає правильний результат;
2) отримавши некоректні дані програма відхиляє їх;
3) програма не зависає і не вилітає, приймаючи як коректні, так і некоректні дані;
4) програма функціонує нормально стільки часу скільки потрібно;
5) програма працює без збоїв і виконує всі необхідні функції в повному обсязі.

1.1 Основні визначення

В цьому розділі викладені основні визначення, відповідно до стандартів, які використовуються як в наукових, так і в прикладних сферах стосовно ПЗ та його якості [12-14].

Помилка (Error) – хибне значення величини на виході системи (або підсистеми), викликане *несправностями* або *збоями*, яке, в свою чергу, може викликати *відмову*.

З точки зору надійності ПЗ помилку можна розглядати, як упущення або неточність, допущені проектувальниками ПЗ, програмістами, аналітиками та тестерами. Наприклад, проектувальник може неправильно зрозуміти завдання, а програміст – неправильно описати змінну та інш.

Несправність, Дефект (Fault) – визнана або передбачувана причина помилки; наслідок відмови деякої системи, що обслуговувала або обслуговує в даний момент часу розглянуту систему. Дефекти також часто називають “багами” (від англ. – “bugs” жучки). Цей термін використовують, якщо вплив дефекту на роботу програми невеликий. Якщо ж помилка пов’язана із специфікаціями або архітектурою програми, то використовують слово “дефект”.

Збій (Malfunction) – прояв несправності, зазвичай в роботі устаткування.

Відмова (Failure) - порушення нормального функціонування системи, повна або часткова втрата працездатності системи (або підсистеми).

Під час виконання програми або роботи всієї системи, тестер, розробник або користувач можуть не отримати очікуваних результатів. В деяких випадках така поведінка – симптом помилки. Досвідчений розробник/тестер завжди зберігає базу даних помилок, з якими він стикався.

Некоректна поведінка може також означати неправильні значення вихідних, неправильний відгук пристрою або неправильне зображення на екрані. В процесі розробки відмови та баги зазвичай виявляються тестерами, а дефекти знаходяться і виправляються самими розробками. Якщо відмова виникла у користувача, звіт про помилку прямує розробнику з метою її усунення.

Несправність у коді коду не завжди веде до відмови. Насправді неправильна частина програми може функціонувати довгий час без прояву яких-небудь недоліків. Проте, за відповідних умов несправність викликає відмову. А саме:

1) вхідні дані програми повинні використовувати частину коду, яка містить несправність;

2) наслідком несправності має стати некоректний внутрішній стан програми;
3) некоректний внутрішній стан програми повинен впливати на вихідні дані так, щоб результат помилки можна було спостерігати.

Якщо при наявності несправності у коді ПЗ відповідає умовам зазначеним вище, то воно вважається ***придатним для тестування (testable)***.

Тестування (***testing***) – це:

- Процес використання ПЗ при умові аналізу або запису отримуваних результатів з метою перевірки (оцінки) деяких властивостей тестованого об'єкту. (*The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*).
- Процес аналізу ПЗ з метою виявлення відмінностей між створеним станом ПЗ та зазначеним у специфікації (що свідчить про прояв помилки) при експериментальній перевірці відповідного пункту вимог. (*The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs)*)
- Контрольоване виконання програми на множині тестових даних і аналіз результатів цього виконання для пошуку помилок.

Тест ***(Test)*** – контрольна задача для перевірки коректності функціонування ПЗ. Групу взаємозв’язаних тестів називають **комплектом тестів** (***test suite***).

Контрольний приклад (test case). Контрольний приклад в сенсі тестування – записи, що відносяться до тесту, а саме, звіти, що містять:

1. *Вхідні дані тесту* – інформація, яку програма отримує із зовнішнього джерела, як то пристрій, інша програма або людина.
2. *Умови виконання* – вимоги для проведення тесту, наприклад, певний стан бази даних або конфігурація пристрою.
3. *Очікувані вихідні дані* – передбачуваний результат роботи коду.

Даний перелік визначає мінімальну необхідну інформацію для контрольного прикладу відповідно до стандартів. Компанія-розробник може визначити додаткову інформацію необхідною для внесення в звіт з метою її повторного використання в майбутньому або надання докладніших даних іншим тестерам і розробникам. Наприклад, мета тесту може бути включена в запис для більшої зрозумілості результатів тесту. До визначення точного формату звіта контрольного приклада слід залучати розробників, тестерів та/або відділ забезпечення якості ПЗ.

Якість (*Quality)* – ступінь відповідності системи, компоненту або процесу заданим вимогам, потребам або очікуванням користувача. З метою визначення добротності системи, компоненту або процесу використовують так звані атрибути якості – характеристики, що відображають дану властивість.

Метрика *(Metrics)* – кількісна міра ступеня наявності *атрибута* системи, компоненту або процесу.

Приведемо декілька прикладів атрибутів якості з коротким описом [15]:

- властивість, що відповідає за безперервність коректного протікання процесу, відноситься до **надійності (reliability)**;

- властивість, що відповідає за постійну готовність системи, відноситься до **готовності (availability)**;
- властивість, що відповідає за відсутність катастрофічних наслідків в системному середовищі, відноситься до **безпеки (safety)**;
- властивість, що відповідає за запобігання несанкціонованому доступу до інформації, відноситься до **конфіденційності (confidentiality)**;
- властивість, що відповідає за відсутність появи в системі невідповідних змін інформації, відноситься до **цілісності (integrity)**;
- властивість, що відповідає за здатність системи піддаватися ремонту і розвитку, відноситься до **ремонтопридатність (maintainability)**.
- властивість, що відповідає за здатність – рівень зусиль, необхідних для навчання, роботи, підготовки вхідних і обробки вихідних даних ПЗ, відноситься до **зручності роботи (usability)**.

Перевірка на валідність(Validation) - процес, що дозволяє визначити, наскільки точно з позицій потенційного користувача деяка модель представляє задану суть реального миру.

Верифікація (Verification) - процес, який дозволяє визначити, що розроблене програмне забезпечення точно реалізує концептуальний опис даної системи. Або, як ще кажуть, процес перевірки відповідності системи заданими стандартами. Верифікація успішна, якщо отримані дані збігаються з очікуваними, заздалегідь визначеними як правильні. Зазначимо, що вона може бути неформальною, тобто тестер визначає успішність на основі своїх знань.

Верифікація та *перевірка на валідність* часто використовуються разом, але мають різні визначення. Різниця в них важлива для тестування ПЗ. Верифікація – це підтвердження того, що продукт відповідає специфікаціям, а перевірка на валідність – вимогам користувача. Може здатися, що визначення вельми схожі, але на прикладі опису проблем орбітального телескопа Хаббла видно різницю.

У квітні 1990 року до космосу було запущено орбітальний телескоп Хаббла, що використовував велике дзеркало для збільшення об'єктів, на які був націлений. Його конструкція вимагала неймовірної точності, а тестування було майже неможливим, оскільки він призначався для космосу і не міг бути встановлений на Землі. У зв'язку з цим, можна було лише перевірити на відповідність специфікаціям. Після проходження таких тестів, телескоп було запущено.

На жаль, після введення його в експлуатацію виявилось, що зображення, які надсилались їм, є розфокусованими. Дослідження показало, що було виготовлено неправильне дзеркало. Не дивлячись на те, що, можливо, це було найбільш точно розраховане дзеркало з коли-небудь створених, а допуск складав не більше 1/20 довжини хвилі видимого світла, воно виявилося дуже плоским по краях. Відхилення від потрібної форми поверхні склало лише 2 мікрометри, але результат виявився катастрофічним — сильна сферична аберація, оптичний дефект, при якому світло, відображене від країв дзеркала, фокусується в точці, відмінній від тієї, в якій фокусується світло, відображене від центру дзеркала. Тобто тестування показало, що дзеркало пройшло верифікацію, але не пройшло перевірку на валідність. У 1993 році експедиція на космічному кораблі відремонтувала телескоп, встановивши коректуючу лінзу для відновлення фокусу. Хоча цей приклад не відноситься до ПЗ,

верифікація та перевірка на валідність також успішно застосовуються і до тестування програм. Отже, сама верифікація не є достатньою. Лише повна перевірка дозволить уникнути проблем, з якими зіткнулися учені у випадку з телескопом Хаббла.

Методи забезпечення якості є техніками, що гарантують досягнення певних показників якості при їх застосуванні.

Методи контролю якості дозволяють переконатися, що певні характеристики якості ПЗ досягнуті. Самі по собі вони не можуть допомогти їх досягненню, вони лише дають змогу визначити, чи вдалося отримати в результаті те, що хотілося, чи ні, а також знайти помилки, дефекти і відхилення від вимог.

Методи контролю якості ПЗ можна класифікувати таким чином:

- Методи та техніки, пов'язані з аналізом властивостей ПЗ під час його роботи. Це, перш за все, всі види тестування, а також вимірювання кількісних показників якості, які можна визначити за наслідками роботи ПЗ, — ефективність за часом й іншими ресурсами, надійність, доступність та ін.
- Методи та техніки визначення показників якості на основі симуляції роботи ПЗ за допомогою моделей різного роду. До цього вигляду відносяться перевірка на моделях (model checking), а також прототипіювання (макетування), що використовується для оцінки якості прийнятих рішень.
- Методи та техніки, націлені на виявлення порушень формалізованих правил побудови початкового коду ПЗ, проектних моделей та документації. До методів такого роду відноситься інспекція коду, що полягає в цілеспрямованому пошуку певних дефектів і порушень вимог в коді на основі набору шаблонів; автоматизовані методи пошуку помилок в коді, не засновані на його виконанні; методи перевірки документації на узгодженість і відповідність стандартам.

- Методи та техніки звичайного або формалізованого аналізу проектної документації і початкового коду для виявлення їх властивостей. До цієї групи відносяться численні методи аналізу архітектури ПЗ, методи формального доказу властивостей ПЗ і формального аналізу ефективності вживаних алгоритмів.

1.2 Рівні і види тестування

В процесі розробки ПЗ тестування ПЗ зазвичай відбувається на декількох рівнях інтеграції: поблочне тестування, перевірка взаємодії (інтеграційне тестування) та системне тестування.

Відповідно до етапів розробки ПЗ прийнято виділяти три фази тестування: модульне, інтеграційне і системне.

Модульне тестування, тестування модуля, або автономне тестування (module testing, unit testing) — контроль окремого програмного модуля, зазвичай в ізольованому середовищі (тобто ізольовано від решти всіх модулів). Під модулем розуміється логічно замкнутий фрагмент програми, який може бути викликаний через його інтерфейс. Модуль перевіряється на відповідність своїм специфікаціям і внутрішню логіку.

Інтеграційне тестування або тестування взаємодій (integration testing) — контроль взаємодії між частинами системи (модулями, компонентами, підсистемами).

Системне тестування або комплексне тестування (system testing) — контроль та/або випробування всього програмного забезпечення, як повної системи, в цільовому середовищі, тобто підтвердження того, що доступ до всіх компонентів системи і взаємодія з ними несуперечливі і передбачені згідно специфікацій системи.

Вочевидь, що фази не є взаємозамінними і, наприклад, проведення модульного тестування не гарантує правильності інтеграційного тестування, бо правильність функціонування окремих компонент не гарантує правильності їх взаємодії, як між собою, так і з системою в цілому.

Альфа- і бета- тестування.

Як правило, перед тим, як вважатися завершеним, програмне забезпечення проходить дві стадії тестування. Перша стадія називається альфа-тестуванням.

Альфа-тестування (alpha testing) — використання майже готової версії продукту (як правило, програмного або апаратного забезпечення) штатними програмістами (розробниками і тестерами) з метою виявлення помилок в його роботі для їх подальшого усунення перед бета-тестуванням.

По закінченню тестування альфи, розробка входить в стадію бети.

Бета-тестування (beta testing) — інтенсивне використання ПЗ з метою виявлення максимальної кількості помилок в його роботі для їх подальшого усунення перед остаточним виходом (випуском) продукту на ринок, до масового споживача.

На відміну від альфа-тестування, що проводиться силами штатних розробників або тестувальників, бета-тестування припускає залучення добровольців з числа звичайних майбутніх користувачів продукту, яким розсилається згадана попередня версія продукту (так звана бета-версія). Такими добровольцями (їх називають бета-тестерами) зазвичай керує цікавість до нового продукту — цікавість, задля задоволення якої вони цілком згодні миритися з можливістю випробувати наслідки ще незнайдених (а тому і невиправлених) помилок.

Крім того, бета-тестування може використовуватися як частина стратегії просування продукту на ринок (наприклад, безкоштовна роздача бета-версій дозволяє привернути широку увагу споживачів до остаточної дорогої версії продукту), а також для отримання попередніх відгуків про нього від широкого круга майбутніх користувачів.

Тестування, що засноване на вимогах (Requirement based testing) **–** тестування кожного припущення з певного документу (документи технічної підтримки, посібники та інша документація користувача).

Регресійне тестування (regression testing, від лати. regressio — рух назад) — спільна назва для всіх видів тестування програмного забезпечення, метою яких є виявлення помилок у вже протестованих ділянках початкового коду. Такі помилки — коли після внесення змін до програми перестає працювати те, що повинне було продовжувати працювати — називають регресійними помилками (regression bugs).

Зазвичай використовувані методи регресійного тестування включають повторні прогони попередніх тестів, а також перевірки, чи не потрапили регресійні помилки в чергову версію в результаті злиття коду.

З досвіду розробки ПЗ відомо, що повторна поява одних і тих самих помилок — випадок доволі поширений. Іноді це відбувається із-за слабкої техніки управління версіями або унаслідок людської помилки при роботі з системою управління версіями. Але настільки ж часто вирішення проблеми буває таким, що «недовго живе»: після наступної зміни в програмі воно перестає працювати. І нарешті, при переписуванні якої-небудь частини коду, часто спливають ті ж помилки, що були в попередній реалізації.

Тому найкращім рішенням є створення тесту на помилку, при її виявленні, з метою його використання при подальших змінах

програми. Хоча регресійне тестування може бути виконане і вручну, але найчастіше це робиться за допомогою спеціалізованих програм, що дозволяють виконувати всі регресійні тести автоматично. У деяких проектах навіть використовуються інструменти для автоматичного прогону регресійних тестів через заданий інтервал часу. Зазвичай це виконується після кожної вдалої компіляції (у невеликих проектах) або щоночі, або кожного тижня.

У термінології тестування поняття «тестування білого ящика» і «тестування чорного ящика» відносяться до того, чи має розробник тестів доступ до початкового коду тестованого ПЗ, або ж тестування виконується через призначений для користувача інтерфейс або прикладний програмний інтерфейс, наданий тестованим модулем.

При ***тестуванні чорного ящика (black-box testing)***, тестер має доступ до ПЗ тільки через ті самі інтерфейси, що і замовник або користувач, або через зовнішні інтерфейси, що дозволяють іншому комп'ютеру або іншому процесу підключитися до системи для тестування.

Можна виділити наступні види тестування чорного ящика:

Тестування за класами еквівалентності (Equivalence class testing). Клас еквівалентності - це множина значень змінної, що, за припущенням, є еквівалентними. Контрольні приклади є еквівалентними, якщо виконується наступна сукупність вимог:

1) вони всі перевіряють один об’єкт;
2) якщо один с них „спіймає” дефект, то інший також;
3) якщо один з них не виявляє дефект, то інший, скоріше за все, також цього не зробить.

Якщо такий клас еквівалентності виявлено, то треба використовувати для тестування лише один з його членів.

Способи вибору таких представників також визначають певний вид тестування:

1) ***Граничне тестування (Boundary testing).*** Граничні значення це найбільші та найменші значення класів еквівалентності. При тестуванні граничних значень тестуються також значення менше за мале, та більше за велике.
2) ***Тестування кращих представників (Best representative testing).*** Тестування значень, що найбільш вірогідно призведуть до виявлення дефекту. Значення завжди можна змінити різними шляхами. Треба покрити всі можливі варіанти.

При ***тестуванні білого ящика (white-box testing, також говорять — прозорого ящика)***, розробник тесту має доступ до початкового коду і може писати код, який пов'язаний з бібліотеками тестованого ПЗ. Це типово для модульного тестування (англ. unit testing), при якому тестуються тільки окремі частини системи. Воно забезпечує те, що компоненти конструкції — працездатні і стійкі, до певного ступеня.

При тестуванні білого ящика розрізняють наступні види:

Тестування логіки (Logic testing). Багато програм мають логіку типу „якщо, то”, тестування логіки тестує можливі сценарії та комбінації.

Тестування станів (State-bases testing). Робота кожної програми - це переходи з одного стану в інший. Тестування станів має на меті уважну перевірку коректності роботи програми у кожному її стані.

Тестування покриття операторів та гілок (Statement and branch coverage). Сто відсоткове покриття має місце коли покриті всі рядки та всі твердження коду. В іншому випадку береться до уваги покриття у процентному відношенні.

Тестування шляхів (Path testing). Тестування набору шляхів (під-шляхів), що проходить програма.

1.3 Техніка тестування

Існує багато прийомів тестування ПЗ. Деякі з них перевершують інші, деякі можна використовувати сукупно з іншими для кращих результатів. Нижче представлений список основних прийомів тестування:

- ***Ручне тестування*** – тести виконуються людьми з наперед складеними, або визначеними для кожного випробування тестовими даними.
- ***Автоматизоване тестування*** – тести виконуються спеціальними інструментами або самостійними процесами і можуть повторюватися багато раз. Тестові дані попередньо вводяться або генеруються.
- ***Регресивне тестування*** – тести, зазвичай автоматизовані, мають на меті виявлення негативного впливу змін в програмі на функції, що пройшли попередні перевірки.
- ***Димове тестування*** – тести, спрямовані на швидку перевірку базової функціональності, з метою виявлення, чи новий білд (версія програми чи її певного модуля) вартий тестування.
- ***Дослідницьке тестування*** – тести, що виконуються за відсутністю специфікацій. Тестер розроблює власну систему тестування, яка базується на накопиченому їм досвіді та оцінює ризики, створюючи сценарії тестування.
- ***Мавпяче тестування*** - тести, що не мають під собою певної системи, «швидка атака» програми тестером.
- ***Стрес-тестування*** – тести призначені для перевірки стійкості програми до надмірного навантаження при нестачі ресурсів

- ***Тестування навантаження*** – тести виконуються при різних рівнях навантаження з метою перевірити поведінку програми та виявити максимально дозволений рівень.
- ***Тестування продуктивності*** – тести виконуються для порівняння поточної продуктивності з розрахунковою.
- ***Тестування інсталяції*** – тести полягають у встановленні програми на різних платформах-комбінаціях та перевірки: чи всі файли було переписано, чи працює програма коректно.
- ***Тестування довгим використанням –*** тести виконуються довготривало, з метою виявлення такого роду помилок, що неможливо виявити при короткому використанні (наприклад, помилки при роботі з динамічним розподілом пам'яті).

2. АКСІОМИ ТЕСТУВАННЯ

З моменту виявлення першого багу, тестування програмного забезпечення пройшло великий шлях. Як всякий новий практичний напрям, воно динамічно розвивалося, не уникнувши і тупикових гілок, невдалих спроб адаптації та перенесення методологій, стандартів і концепцій з вже існуючих областей. Додатковою особливістю цього процесу стала залежність тестування від власне програмного забезпечення, чиї технології, методи та інструменти самі переживають період стрімкого й інтенсивного вдосконалення. Також слід відзначити, що не маючи за спиною багатого досвіду теоретичних досліджень, система забезпечення якості ПЗ, а услід за нею і тестування, протягом довгого часу обростали всілякими міфами і попадали під вплив різних ідейних течій. Мета даного розділу – розвіяти деякі з ілюзій, пов’язаних з тестуванням.

Неможливо повністю протестувати програму

Початківець у сфері тестування може вважати, що можна обробити ПЗ повністю протестувавши його, знайшовши всі помилки, і підсумувавши, що ПЗ ідеальне. Нажаль, це неможливо, навіть для найпростіших програм, через наступні чотири ключові причини:

1. Кількість можливих вхідних даних дуже велика.
2. Кількість можливих результатів дуже велика.
3. Кількість проходів по ПЗ дуже велика.
4. Специфікація ПЗ суб'єктивна. Можна сказати, що помилка – це зовсім не помилка, так і було задумано.

Підсумувавши ці причини, отримуємо набір умов тестування, який занадто великий, щоб його здолати. Ілюстративним прикладом може служити калькулятор Windows.

Приклад: Нехай поставлене завдання на тестування калькулятору Microsoft Windows. Почнемо з операції додавання.

Крок перший: 1+0=? Одержуєте відповідь 1. Коректно

Крок другий: 1+1=? Одержали 2. Коректно

Скільки таких кроків потрібно?

Калькулятор приймає 32- х розрядне число, отже, необхідно перевірити всі можливі варіанти до 1+99999999999999999999999999999999 =.

Закінчивши цю послідовність, треба перейти до 2+0=, 2+1=, 2+2= та інш. Останнім варіантом буде: 99999999999999999999999999999999+99999999999999999999999999999999=

Далі необхідно перевірити коректність додавання дробі: 1.0+0.1,1.0+0.2, та інш.

Після перевірки того, що коректні дані додаються правильно, залишається ввести некоректні дані. Це дасть змогу переконатися, що вони грамотно оброблюються. Нагадаємо, що можна використовувати не лише цифри на екрані, але й набирати символи з клавіатури. Важливо перевірити можливість введення 1+a, z+1, 1a1+2?>2,.... Скільки буде таких варіантів?

Виправлення введення також повинно бути протестовано. Калькулятор Windows дозволяє використовувати Backspace і Delete, отже, належить перевірити і їх.

Kbackspace>2+2 повинно дорівнювати 4.

Всі вищезазначені перевірки мають бути повторені з використанням клавіші Backspace для кожного введення, для кожних двох введень і так далі.

Після закінчення цього етапу можна перейти до додавання трьох чисел, потім чотирьох і т. д.

Це приклади лише для операції додавання, а залишаються ще різниця, множення, ділення, відсоток, корінь квадратний та інш.

Вочевидь, що через кількість контрольних прикладів неможливо завершити повне тестування навіть такої програми як калькулятор.

Перейдемо до розгляду наступної аксіоми.

Тестування – це процес, що містить ризики.

Попередній приклад показав, що перебрати всі варіанти неможливо, треба чимось нехтувати. Якщо приймається рішення не тестувати всі можливі сценарії, то вибирається деякий ризик. У прикладі з калькулятором, що буде, якщо вирішити не перевіряти 1024+1024=2048? Існує ймовірність того, що програміст зробив помилку, яка впливає саме на цю ситуацію. Якщо не протестувати її, користувач рано чи пізно з нею стикнеться. Ця помилка може коштувати дуже дорого, адже вона буде знайдена, коли ПЗ вже знаходиться в експлуатації, а тому на вилучення та заміну версії, що містить баг, мають бути витрачені значні зусилля.

Таким чином, маємо протиріччя – неможливо протестувати все, проте якщо не протестувати, то виникає ймовірність пропущення помилок. Перед розробником стоять суперечливі цілі - продукт повинен бути реалізований, отже, необхідно закінчити тестування, але якщо закінчити його занадто швидко, то залишаться не протестовані частини.

Тестеру необхідно навчитися скорочувати величезну область всіх можливих тестів до керованого набору та приймати, враховуючий ризик, розумні рішення: що важливо для тестування, а що ні.

На рис. 2 проілюстровано залежність між обсягом тестування та якістю проекту у сенсі кількості знайдених помилок. Якщо

спробувати протестувати абсолютно все, ціна різко зросте, а кількість пропущених помилок спаде до нуля.

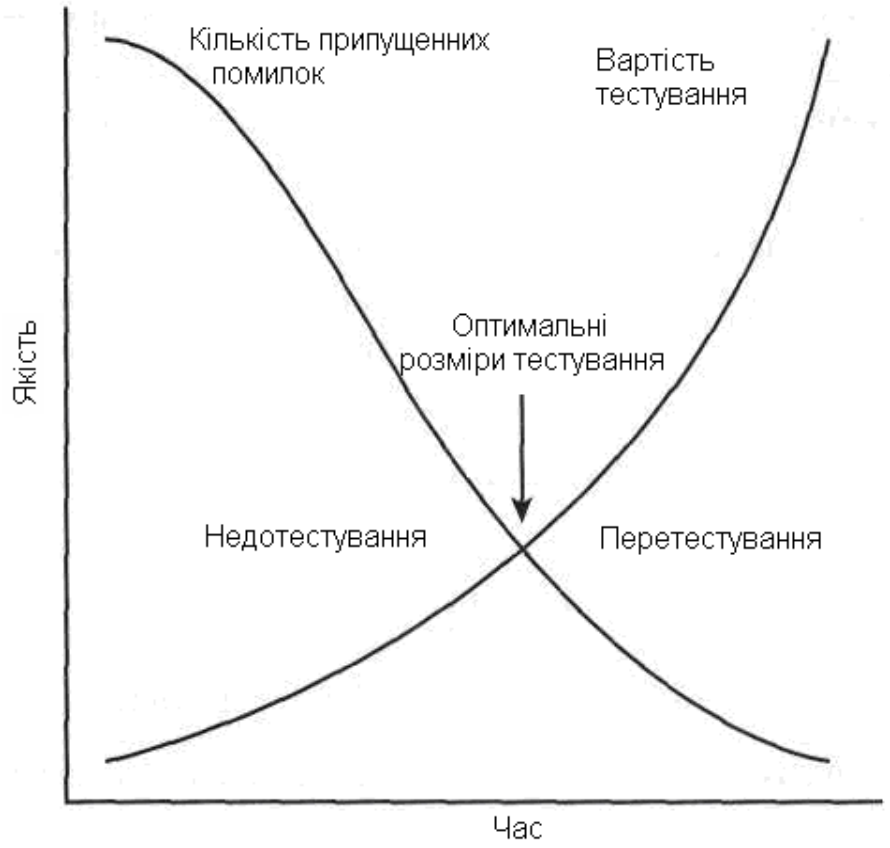

Рис 2. Кожен проект має оптимальний обсяг тестування

Якщо сильно скоротити тестування або приймати хибні рішення відносно того, що саме тестувати, ціна зменшиться, але залишиться велика кількість помилок . Мета - це знайти оптимальний обсяг тестувань.

Тестування не може показати, що помилок немає.

Ця аксіома була висунута професором Е. Дейкстрой в 1972 [11]. Проілюструємо її.

Уявіть роботу винищувача, що шукає в будинку жуків. Він перевіряє будинок і знаходить їх - можливо живих, мертвих або навіть гніздо. Можна спокійно стверджувати, що в будинку є жуки.

Потім винищувач відвідує інший будинок, та не знаходить в ньому присутності жуків. Він дивиться у всіх передбачуваних місцях, але не виявляє ніяких ознак навали. Можливо, буде знайдено кілька мертвих жуків або старих гнізд, але нема нічого, що могло б свідчити про присутність живих жуків. Чи можна однозначно, впевнено заявити, що в будинку жуків немає? Все, що можна сказати – це те, що пошук не виявив жуків. І поки не буде розібрано будинок на частині аж до фундаменту, не можна бути впевненими в тому, що жуків просто випадково не помітили.

Тестування ПЗ дуже схоже на пошук жуків. Воно може показати наявність жуків, але не може показати, що їх немає. Можна провести тестування, знайти й доповісти про помилки, але не можна стверджувати, що їх більше немає. Можна тільки продовжити тестування й, по можливості, знайти інші помилки.

Чим більше помилок знаходить тестер, тим більше їх існує

Існує багато подібностей між реальними жуками й несправностями (bugs) у ПЗ. І ті, і інші мають тенденцію з'являтися групами - якщо була помічена одна, імовірно поблизу є ще. Часто тестер досить довго не може нічого знайти. Потім він раптом знаходить одну помилку, потім дуже швидко наступну й наступну. От кілька причин цього:

- Як і в усіх нас, у програмістів бувають невдалі дні. Код, написаний в один день, може бути ідеальним; код написаний в

іншій - недбалим. Одна помилка може бути маячком, що показує - тут поруч є ще.

- Програмісти часто роблять ті самі помилки. В усіх є звички. Програміст, схильний до певних помилок, буде часто повторювати їх.
- Деякі помилки дійсно тільки вершина айсбергу. Дуже часто проект або архітектура ПЗ мають фундаментальні проблеми. Тестер знаходить помилки, які на перший погляд здаються не зв'язаними, але, в остаточному підсумку, указують на одну серйозну первинну причину.

Важливо відзначити, що зворотне твердження до "помилки випливають за помилками" також правдиво. Якщо тестер, попри всі зусилля, не зміг знайти помилки, це, з великою ймовірністю, значить, що ПЗ було акуратно написане, у ньому міститься всього кілька помилок, які можна знайти.

Парадокс пестицидів

В 1990 році Борис Бейзер у своїй книзі "Техніки тестування програмного забезпечення" запропонував термін парадокс пестицидів - чим більше тестер тестує ПЗ, тим більше воно стає невразливим до тестувань [16]. Аналогічне відбувається з комахами при використанні пестицидів (Рис. 3). Якщо використовувати ті самі пестициди, у комах виробляється захист, і пестициди більше не діють.

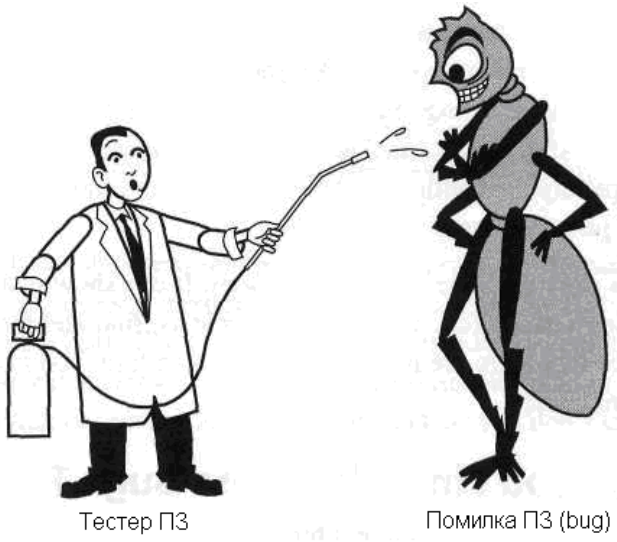

Рис. 3. ПЗ під дією однакових повторюваних тестів стає стійким до них.

Для подолання парадокса пестицидів, тестери ПЗ повинні писати нові, різноманітні тести, що дозволить знаходити більше помилок.

Не всі знайдені помилки будуть виправлені

Одна із реалій тестування ПЗ полягає в тому, що, навіть після наполегливої роботи, не всі помилки будуть виправлені. Це не означає, що тестер не досяг своєї мети, або що команда в цілому випустила поганий продукт. Іноді треба знаходити компроміс та йти на ризик, приймаючи рішення щодо того виправляти помилку, чи ні. Наведемо деякі причини, з яких помилка може бути не виправленою:

- **Недостатньо часу**. У кожному проекті завжди є багато специфікацій та нюансів (іноді занадто багато, що б написати до них код і протестувати його у зазначений час) і не досить можливостей, що б закінчити їх усі.

- ***Це насправді не помилка.*** Часто від розробників можна почути: "Це не помилка, це - властивість!" Це незвично, але помилки тестування або зміни у специфікації можуть призвести до того, що несправності будуть залишені, як властивості.

- ***Занадто ризиковано виправляти.*** Нажаль, саме так буває дуже часто. ПЗ іноді схоже на спагетті: виправлення однієї помилки може спричинити виникнення нових. Під тиском реалізації та щільного графіка може бути занадто ризиковано змінювати ПЗ. Тому краще залишити відому помилку, чим ризикувати створити нову, невідому.

- ***Це просто не варто виправляти.*** Можливо, це звучить грубо, але це реальність. Помилки, які виникають нерегулярно або в мало використовуваних функціях можуть бути опущені. Причина цього – бізнес рішення, що базуються на ризику.

Для прийняття рішень звичайно необхідні тестери ПЗ, керівник проекту та програмісти. Кожний піклується про своє бачення майбутнього помилки, має свої дані та думку - чому треба або не треба її виправляти.

Що відбувається, коли приймається неправильне рішення?

Можна згадати рішення Intel Pentium, коли тестери Intel знайшли помилку при обробці чисел з плаваючою точкою раніше, ніж чип був випущений, але команда проектувальників вважала, що вона є незначною і її не обов'язково виправляти. У них був дуже тісний графік, тому вони вирішили продовжувати проект за планом, а знайдену помилку виправити в наступних версіях. На жаль, помилка була виявлена користувачами, і компанія понесла великі фінансові збитки.

Іноді складно сказати чи є помилка помилкою

Якщо в ПЗ є проблема, але її не було знайдено ні програмістами, тестерами, ні навіть користувачами – чи помилка це?

Якщо задати це питання групі тестерів, то можна побачити вражаючу дискусію. У кожного з них буде своя власна думка й кожний зможе добре та яскраво її аргументувати. Проблема в тому,

що певної відповіді на це питання не існує. Вона полягає у тому, що на певний час краще для тестерів та команди розроблювачів.

Твердження, що програмне забезпечення робить або не робить "чогось" припускає, що воно було запущено, "щось" протестовано або ж недолік "чогось" був очевидний. Пізніше не можливо буде доповісти про те, чого не видно та не можна відтворити. Визнавайте помилку помилкою, тільки якщо вона спостерігається.

Про це можна подумати й по-іншому. Зовсім не дивно, що дві людини мають зовсім різні думки про якість ПЗ. Один може сказати, що програма містить жахливо багато помилок, інший - що вона ідеальна. Як обоє можуть бути праві? Відповідь проста - один користується програмою так, що з'являються помилки, а інший – що ні.

Специфікація розробки ніколи не завершується

Спочатку необхідно розібратися з терміном: специфікація розробки.

Специфікація розробки, іноді називана просто специфікацією (англ. product specification, product spec, spec) – це домовленість між членами команди розроблювачів ПЗ. Вона визначає продукт, що вони створюють, деталізує, яким він буде, як буде працювати, що буде робити й чого не буде. Ця домовленість може варіюватися: від простої усної згоди до формалізованого записаного документа.

У розробників ПЗ є проблема. Індустрія розвивається так швидко, що найпередовіші розробки минулого року в цьому вже є застарілими. ПЗ стає більшим, складнішим й містить у собі все більше функцій, отже, як результат, подовжуються й подовжуються списки для його розробки.

Не існує іншого шляху реагування на настільки швидкі зміни. Припустимо, що продукт має закриту, закінчену і не підлягаючу зміні

специфікацію. На середині дворічного циклу виробництва продукту А, головний супротивник випускає дуже схожий продукт В з декількома корисними властивостями, яких немає в А. Що робити далі с продуктом А? Продовжувати роботу за специфікацією й випустити через рік другосортний продукт? Або перегрупувати команду, переписати специфікацію, і працювати над виправленою розробкою? У більшості випадків, розумно останнє.

Тестер ПЗ повинен засвоїти, що специфікація буде мінятися. Властивості й функції будуть додаватися, не заважаючи на те, що спочатку вони не мали бути протестовані. Вони будуть змінюватися або взагалі видалятися, хоча вже були протестовані та позбавлені частини помилок.

Тестери ПЗ не самі популярні члени команди розробників

Мета тестера ПЗ – якомога раніше знаходити помилки і робити так, щоб вони були виправлені.

Тобто робота тестера полягає в тому, що він змушений перевіряти та контролювати своїх колег, знаходити їхні проблеми і оголошувати їх. От кілька варіантів, як підтримувати мир із членами команди:

- ***Знаходити помилки як можна раніше.*** Набагато меншим збитком для всіх і набагато більшим плюсом тестеру буде, якщо він знайде серйозну помилку за три місяці, а не за день, до випуску програми.
- ***Стримувати захват.*** Добре, якщо тестер дуже полюбляє свою роботу та приходить у захват, коли знаходить серйозну помилку. Але, якщо він увірветься до кімнати програмістів з яскравою посмішкою й скажіть їм, що знайшов в їхній частині коду найжахливішу помилку із всіх, вони не будуть дуже щасливі.

- ***Приносити не тільки погані новини.*** Якщо завжди казати погане, то згодом люди почнуть уникати тестера, щоб не отримувати поганих звісток. Тому, якщо він знайшов шматочок коду без помилок, то буде краще сказати усім про це. Зрідка можна заходити до програмістів взагалі просто задля того, щоб побалакати.

3. МІСЦЕ ТЕСТУВАННЯ В ЦИКЛІ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

В попередніх розділах йшлося про помилки програмного забезпечення (ПЗ). Дуже часто називаючи усі проблеми ПЗ «помилками» (bugs), тепер треба розкрити суть цього поняття.

Повернемося до визначення некоректної роботи ПЗ:

1) ПЗ не робить чогось, що, відповідно до специфікації, воно повинне робити.
2) ПЗ робить щось, що, відповідно до специфікації, воно не повинне робити.
3) ПЗ робить щось, що не згадується в специфікації.
4) ПЗ не робить чогось, що не згадується в його специфікації, але повинне.
5) ПЗ складно зрозуміти, важко використовувати, воно повільне, або - на думку тестерів ПЗ - буде сприйнято кінцевими користувачами, як явно не правильне.

Проілюструємо роботу цих правил на прикладі калькулятора:

Нехай у специфікації калькулятора зазначено, що він має правильно виконувати додавання, віднімання, множення та ділення. Якщо тестер, одержавши калькулятор, натискає кнопку «+», і нічого не відбувається, то це помилка відповідно до правила №1. Якщо надана відповідь буде неправильною, то це теж буде помилка №1.

Специфікація може містити вимогу, що калькулятор ніколи не повинен зависати. Якщо тестер щось увів, і калькулятор перестав реагувати на всі подальші дії, це помилка відповідно до пункту №2.

Припустимо, тестер одержав калькулятор на тестування й виявив, що крім додавання, віднімання, множення й ділення, він також містить квадратний корінь. Ніде в специфікації цього не було - честолюбний програміст просто додав його, оскільки вирішив, що це

буде корисна функція. Це не функція - це скоріш за все помилка відповідно до правила №3.

Правило №4, можливо, звучить дивно з подвійним запереченням, але його завдання – вловити нюанси, які були упущені в специфікації. Тестер почав тестувати калькулятор і виявив, що коли батарея сідає, він перестає рахувати правильно. Ніхто ніколи не замислювався, як повинен поводитися калькулятор у таких випадках. Поганим рішенням була б розробка ПЗ з розрахунком на те, що батарея повинна завжди бути повністю зарядженою. Тестер очікував, що зможе продовжити свою роботу, доти батарея не сяде остаточно, або хоча б доти там не залишиться зовсім трошки заряду. Калькулятор рахує не правильно з батареєю, що розряджається, але ніде не записано, як він повинен поводитися в такій ситуації. Це помилка відповідно до правила №4.

Правило №5 дуже узагальнене. Тестер – перша людина, що користується програмою. Якщо це будете не він, то першими користувачами стануть покупці. Якщо тестер знайде щось, що здасться йому не правильним, не важливо чому, - це помилка. У випадку з калькулятором, можливо, він знайде, що кнопки замалі, через що натискати їх незручно. Можливо, при яскравому освітленні складно розібрати, що на екрані. Все це помилки відносно правила №5.

Зауважимо, що кожна людина, яка використовує ПЗ, буде мати різні сподівання і свою думку що до того, як воно повинно працювати. Це неможливо - написати ПЗ, яке всі користувачі назвуть ідеальним. Тестер ПЗ повинен завжди пам'ятати про це, застосовуючи правило №5 у своїй роботі.

3.1 Чому виникають помилки?

Треба зауважити, що більшість помилок виникають не через помилки програмістів. Аналіз багатої кількості розробок виявив: головна причина виникнення помилок у ПЗ – це специфікація (Рис. 4).

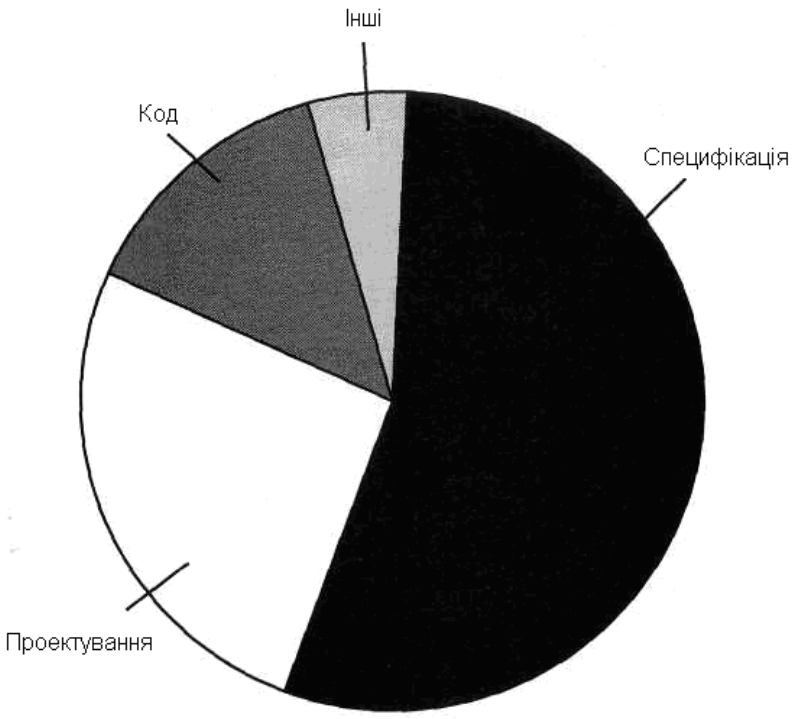

Рис. 4. Помилки виникають на багатьох етапах розробки, але найбільше – на етапі специфікації.

Є кілька причин, які роблять специфікацію основним "постачальником" помилок. Інколи її просто не існує, інколи – вона неповна, або занадто часто змінюється. Написання та аналіз специфікацій необхідна та найголовніша частина циклу розробки ПЗ. Якщо цей етап пропущено чи здійснено некоректно, помилки будуть виникати.

Наступним джерелом виникнення помилок є проектування. Неправильно розроблена архітектура ПЗ через поспіх,

непродуманість, відсутність досвіду, непослідовність та ін. може мати величезні наслідки

Помилки в коді виникають через складність ПЗ, бідність документації, високу щільність графіка, або випадково. Важливо відзначити, що у більшості випадків помилки, які виникли на поверхні, указують на проблеми специфікації та проектування.

Остання категорія містить у собі все, що залишилось. Деякі помилки можуть виявитися помилками, що з якихось причин були прийняті за них, але ними не були. Можуть бути повторювані помилки, скопійовані із одного джерела. Деякі помилки можуть вказувати на неправильне тестування. В остаточному підсумку, ці помилки зазвичай становлять настільки малий відсоток від інших, що не варто особливо хвилюватися з цього приводу.

3.2 Ціна помилок

ПЗ не виникає миттєво – в його основі лежать планування й методики процесу розробки. Починаючи з виникнення ідеї, далі в процесі планування, програмування й тестування та закінчуючи використанням цього програмного забезпечення користувачами, потенційно можуть бути знайдені помилки.

На рис. 5 проілюстровано, скільки коштує виправити помилку, знайдену на різних етапах.

Ціна логарифмічна - тому, із часом вона зростає десятикратно. Помилка, виправлена на самому початку, тобто на етапі специфікації, нічого не коштує, або 10 центів, як у прикладі. Та ж помилка, не знайдена до написання коду й тестування ПЗ, може коштувати від $1 до $10. А якщо її знайдуть користувачі, то вартість виправлення легко перевищить $100.

Наглядним прикладом є випадок з компанією Дісней, а саме її грою «Король лев»[17].

Восени 1994 року, компанія Дісней реалізувала на компакт-дисках свою першу мультимедійну гру для дітей «Король Лев, анімована збірка розповідей» (The Lion King Animated Storybook). Не дивлячись на те, що багато інших компаній займалися продажем дитячих програм роками, це була перша розробка Діснея, що поступала на ринок, тому вона була відмінно представлена і розрекламована. Продажі були величезні – це була «гра, яку варто купити для дітей на свята». Проте те, що трапилося пізніше, було колосальним розгромом. 26 грудня, тобто на наступний день після Різдва, телефони підтримки покупців Діснея захлинулися від дзвінків розлючених батьків з дітьми, які не могли змусити програму працювати.

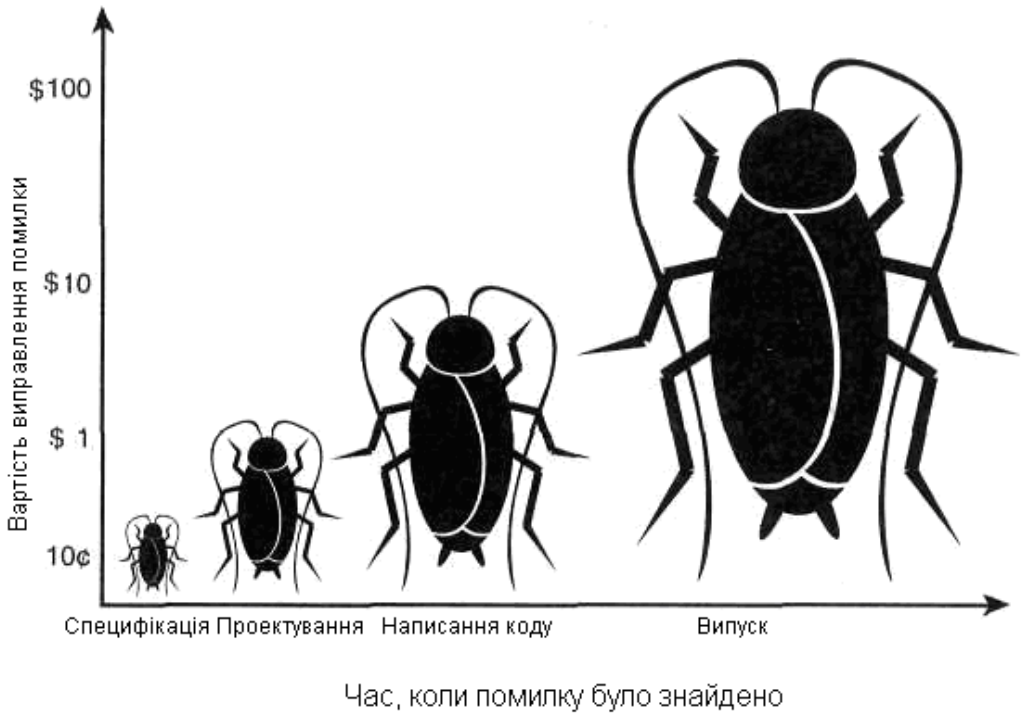

Рис. 5. Ціна виправлення помилки.

Виявилось, що Дісней не протестував належним чином свою програму на різних версіях ПК, представлених на ринку. Програма працювала всього в декількох системах – подібних тій, яку розробники Діснея використовували для створення гри, - але не в найбільш поширених, які були у більшості покупців.

Якби на етапі аналізу вимог було досліджено, які ПК користуються попитом, та вписано в специфікацію, що гра повинна розроблятися і тестуватися на їх конфігурації, помилки б не виникло. Якщо це не було зроблено на етапі специфікації, тестери мусили б зібрати всі популярні моделі ПК і протестувати на них гру. Тоді вони б знайшли помилку, але вона, безумовно, коштувала б дорожче, тому що ПЗ було вже написано, налагоджене й протестовано. Команда розроблювачів також могла зробити бета тест і розіслати першу версію ПЗ невеликій кількості користувачів. Ці користувачі, обрані представляти величезний ринок, із задоволенням знайшли б помилку. Як виявилося, незважаючи ні на що, помилка лишилась непоміченою, а багато тисяч дисків були створені й продані. Компанія Дісней, в решті решт, сплачувала витрати на телефонну підтримку користувачів, повернення продукту, заміну дисків та інш. Отже, можна побачити, як легко втратити весь дохід від розробки, якщо серйозна помилка дійде до користувачів.

4. ПРИНЦИПИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Принципи і правила грають важливу роль у всіх інженерних дисциплінах. Принципи тестування важливі для спеціалістів/ інженерів-тестерів, оскільки вони забезпечують основу для надбання знань і навичок в цій сфері.

Слово «принцип» має декілька значень:

- *загальний або фундаментальний закон, доктрина або допущення;*
- *правило або кодекс поведінки;*
- *закони або факти, які складають основу роботи штучного пристрою.*

Застосовуючи ці визначення до сфери розробки ПЗ, можна сказати, що її принципи лежать в законах, правилах і доктринах, що відносяться до систем ПЗ, способів їх побудов та їх поведінці. У сфері ПЗ до принципів також можуть бути віднесені правила і кодекси поведінки професіоналів, що проектують, розробляють, тестують та управляють системами ПЗ. Тестування, як етап розробки ПЗ, також має особливий набор принципів необхідних для тестера. Вони вказують, як перевіряти системи ПЗ, і визначають правила поведінки тестера-професіонала. Гленфорд Майєрс в своїй книзі «Мистецтво тестування ПЗ»[6,7] виділив список обов'язкових принципів. Вони описані нижче. Принципи, що представлені в таблиці 1 праворуч, взяті з оригінального списку Майєрса. Ліворуч представлені видозмінені правила відповідно до еволюції тестування від мистецтва до одного з процесів інженерної дисципліни, що безпосередньо відноситься до якості [17]. Треба зауважити, що описані нижче принципи відносяться лише до тестування, заснованого на виконанні ПЗ. Принципи, які відносяться до переглядів, сертифікації, доказу

коректності в рамках тесту не розглядаються. Більшість з приведених принципів може здаватися очевидною, тим не менше, часто про них забувають.

Таблиця 1. Принципи тестування програмного забезпечення

Нові принципи	**Оригінальні принципи**
Принцип 1: Тестування – це процес випробування програми за допомогою вибраного набору варіантів тесту з метою: 1) виявити помилки, 2) оцінити якість.	**Принцип 1:** Необхідна складова контрольного прикладу – визначення очікуваного результату та вихідних даних.
Принцип 2: Якщо мета тесту – визначити помилки, то якісний варіант тесту той, що найімовірніше виявить незнайдену досі помилку.	**Принцип 2:** Програмісту слід уникати спроб тестувати свою власну програму.
Принцип 3: Результати тесту повинні бути ретельно перевірені.	**Принцип 3:** Компанії, яка займається розробкою коду, не слід тестувати свої власні програми.
Принцип 4: Контрольний приклад повинен містити очікуваний результат та вихідні дані.	**Принцип 4:** Треба ретельно досліджувати результати кожного тесту.
Принцип 5: Тести повинні розроблятися як для коректних вхідних даних, так і для некоректних.	**Принцип 5:** Тести повинні писатися як для некоректних і неочікуваних вхідних даних, так і для коректних та очікуваних.
Принцип 6: Існування більшого числа помилок пропорційне кількості знайдених помилок в компоненті.	**Принцип 6:** Дослідження програми на правильність функціонування – лише половина роботи. Друга половина – виявити зайву функціональність програми.
Принцип 7: Тестування повинне	**Принцип 7:** Краще уникати одноразових

проводитися групою людей, незалежною від розробників.	варіантів тестів, якщо сама програма не одноразового використання.
Принцип 8: Тести повинні передбачати багаторазове виконання.	**Принцип 8:** Не слід планувати тестування, негласно вважаючи, що помилок немає.
Принцип 9: Тестування слід планувати.	**Принцип 9:** Існування більшого числа дефектів пропорційно кількості знайдених дефектів в компоненті.
Принцип 10: Слід включати процес тестування в життєвий цикл ПЗ.	**Принцип 10:** Тестування, це завдання, що вимагає творчості та розумової праці.
Принцип 11: Тестування, це завдання, що вимагає творчості та розумової праці.	

Проаналізуємо наведені принципи:

> **Принцип 1.** Тестування – це процес випробування програми за допомогою вибраного набору варіантів тесту з метою
> 1) виявити помилки,
> 2) оцінити якість.

В оригіналі:

> **Принцип 6:** Дослідження програми на правильність функціонування – лише половина роботи. Друга половина – виявити зайву функціональність програми.

Цей очевидний принцип – одна з помилок, що найбільш часто зустрічаються, в тестуванні. Знову таки, це наслідок людської психології. Якщо очікуваний результат варіанту тесту не представлений, існує ймовірність, що правдоподібний, проте некоректний результат вважатимуть правильним через принцип «очі бачать те, що хочуть». Іншими словами, незважаючи на чітке визначення тестування, підсвідомо тестер бажає побачити правильний результат. Один із способів боротьби з цим – уважно перевіряти вихідні дані, заздалегідь точно сформулювавши

очікуваний результат. Таким чином, варіант тесту повинен складатися з двох частин:

- *Опис вхідних даних програми.*
- *Точний опис правильних вихідних даних при зазначених вхідних.*

Розробники ПЗ набули значних навичок в запобіганні та знищенні дефектів та помилок. Тим не менш, останні трапляються і негативно впливають на якість програм. Для виявлення наявності дефектів до того, як програма буде випущена на ринок, необхідні тестери. Цей принцип підтверджує той факт, що тестування засноване на виконанні програми. Це і визначає відмінність тестування від відладки, метою якої є знаходження місця дефекту і його усунення. Термін «дефекти» використовується в цьому і наступних принципах для позначення будь-яких відхилень в програмі, які негативно впливають на її функціональність, продуктивність, надійність і/або інші атрибути її якості.

Тестування може бути розглянуте як «динамічний процес виконання програми із цінними вхідними даними». Такий підхід, разом з визначенням тестування, наданим вище, припускає, що тестування не лише виявляє дефекти, але і використовується для оцінки якості ПЗ. У такому разі тестер виконує програму, використовуючи контрольні приклади, для оцінки таких характеристик, як надійність, зручність використання, рівень продуктивності та інші. Результати тесту дозволяють порівняти поточну якість з рівнем, описаним в документації. Неможливість досягти необхідної якості або будь-які відхилення обов'язково повинні бути розглянуті.

Перейдемо до розгляду наступної пари принципів.

Принцип 2. Якщо мета тесту – визначити дефекти, то якісний варіант тесту той, що найімовірніше виявить незнайдений досі дефект.

В оригіналі:

Принцип 8: Не слід планувати тестування, негласно вважаючи, що помилок немає.

Принцип 2 підтримує акуратну розробку тесту та задає критерій оцінки контрольного прикладу і ефективності тесту в тих випадках, коли мета – виявлення дефектів. Тестеру потрібно визначити ціль кожного контрольного прикладу, тобто виділити тип шуканого дефекту. Таким чином, тестер виконує перевірки аналогічно вченому, який проводить експеримент. У вченого існує гіпотеза, яку він повинен підтвердити або спростувати експериментальним шляхом. У тестера гіпотеза пов'язана з можливою наявністю конкретних типів дефекту. Мета тесту – довести/спростувати тезу, тобто визначити наявність/відсутність конкретного дефекту. На основі тези обираються вхідні і визначаються правильні вихідні дані тесту, після чого він проводиться. Результати аналізуються для доказу/спростування тези. Потрібно розуміти, що на процес тестування витрачається багато ресурсів, включаючи ресурси для розробки контрольних прикладів, їх виконання, запису і аналізу результатів. За допомогою ретельної розробки тесту відповідно до принципу 2 тестер може добитися мінімальної витрати ресурсів.

Розглянемо наступний принцип.

Принцип 3. Результати тесту повинні бути ретельно перевірені.

В оригіналі:

Принцип 4: Треба ретельно досліджувати результати кожного тесту.

Тестеру необхідно ретельно досліджувати й аналізувати результати тесту. Інакше можливі несприятливі наслідки. Наприклад:

- *Може бути пропущено критичну помилку, при цьому тест вважатиметься пройденим, хоча програма його провалила. Тестування може продовжитися з урахуванням неправильного результату, і, навіть, якщо дефект буде виявлено на пізнішій стадії тесту, його усунення виявиться важчим та більш ресурсоємним.*
- *Виникне припущення про критичну помилку, якої не існує. У такому разі тест вважатиметься проваленим. Значну кількість часу і ресурсів буде витрачено на виявлення неіснуючого дефекту. І лише повторна перевірка покаже, що помилки не було.*
- *Вихідні дані тесту на якість будуть інтерпретовані невірно, що може призвести до непотрібної переробки або пропуску критичної помилки..*

Отже, без відповідної уваги до результатів процес тестування вимагатиме більше фінансових та часових ресурсів.

Наступний принцип формулюється наступним чином:

Принцип 4. Контрольний приклад повинен містити очікуваний результат та вихідні дані.

В оригіналі:

Принцип 1. Необхідна складова контрольного прикладу – визначення очікуваного результату та вихідних даних..

Зазвичай зрозуміло, що вхідні дані – це складова частина контрольного прикладу. Проте, доки відсутній докладний опис очікуваних вихідних даних або результатів, наприклад, значення конкретної змінної або підсвічування потрібної кнопки на панелі, контрольний приклад не може вважатися повноцінним. Очікувані вихідні дані дозволяють тестеру визначити:

1) чи виявлено дефект;

2) чи пройдено/провалено тест.

Дуже важливо чітко описувати вихідні дані, щоб не витрачати час на уточнення дрібниць. Визначення конкретних вхідних і вихідних даних повинно бути частиною процесу розробки тесту.

В разі тестування на оцінку якості, має сенс описати цільовий рівень якості кількісними даними в спеціальному документі з вимогами. Це дасть тестерам змогу порівнювати поточну якість програми з тим рівнем, що вимагається.

Наступний принцип:

> **Принцип 5.** Тести повинні розроблятися як для коректних вхідних даних, так і для некоректних.

В оригіналі:

> **Принцип 5:** Тести повинні писатися як для некоректних і неочікуваних вхідних даних, так і для коректних та очікуваних.

Тестер не повинен вважати, що програма на тесті повинна забезпечуватися лише коректними вхідними даними. З деяких причин це не так. Наприклад, користувачі можуть не бути обізнані про те, які дані можна вводити. До того ж, вони часто роблять друкарські помилки навіть за наявності повних/коректних вхідних даних. Некоректні вхідні дані також можуть бути наслідком несправності ПЗ чи системи в цілому. Використання контрольних прикладів із некоректними вхідними даними корисно для виявлення дефектів, оскільки у такому разі код виконується нестандартним способом і примушує програму поводитися несподівано. Некоректні вхідні дані також допомагають розробникам і тестерам оцінити надійність програми, її здатність відновлюватися (у разі помилкового введення).

Зауважимо, що представлена пара принципів підтверджує необхідність залучення до тестування незалежної групи тестерів, яка згадується у принципі 7, бо розробник програмного компоненту може бути схильний вибирати лише коректні вхідні дані для демонстрації

працездатності програми. Незалежний тестер схильний обирати також і некоректні вхідні дані.

Перейдемо до розгляду наступної групи принципів:

Принцип 6. Існування більшого числа дефектів пропорційне кількості знайдених дефектів в компоненті.

В оригіналі:

Принцип 9: Існування більшого числа дефектів пропорційне кількості знайдених дефектів в компоненті.

Принцип стверджує, що чим більше дефектів вже знайдено в компоненті, тим імовірніше виявлення додаткових дефектів при подальшому тестуванні. Наприклад, якщо тестери знайшли 20 дефектів в компоненті А і 3 в компоненті В, імовірність існування додаткових дефектів в А більше, ніж у В. Дефекти часто виявляються в класах і в неякісно написаному коді з високим рівнем складності. За наявності таких компонентів розробники і тестери повинні вирішити, чи кинути поточну версію і переробити її або виділяти більше ресурсів на тестування доки компонент не відповідатиме вимогам якості. Такі випадки критичні для важливих компонентів, наприклад тих, що відповідають за безпеку.

Наступні принципи:

Принцип 7. Тестування повинне проводитися групою людей, незалежною від розробників.

В оригіналі цьому принципу відповідало два:

Принцип 2. Програмісту слід уникати спроб тестувати свою власну програму.

Принцип 3: Компанії, що займається розробкою коду, не слід тестувати свої власні програми.

Кожен письменник знає – або повинен знати – що намагатися виправляти або коректувати власну роботу – погана ідея. Ви пам’ятаєте, що мали на увазі в тій або іншій частині і можете не помітити, що вона містить ще щось. І вже точно ви не хочете шукати в своїй роботі помилки. Цей підхід стосується і авторів програм.

Більш того, тестування має на меті іншу задачу ніж написання коду – в його основі лежить «деструктивний» підхід: треба довести, що програма не працює. А програмісту, який написав програму, дуже складно переключитися на її випробування, що носять «руйнівний» характер.

Багато господарів знають, що видалення шпалер зі стін (руйнівний процес) – нелегка справа, яка перетворюється на муку, якщо шпалери були поклеєні власноруч. Аналогічно, більшість програмістів не в змозі ефективно тестувати власні програми, оскільки вони не можуть зосередитися на спробах знайти помилки. До того ж, програміст може підсвідомо уникати знаходження помилок через страх перед покаранням з боку керівництва, клієнта або власника програми, що розробляється. Додатково з цими психологічними перешкодами існує і ще одна значна проблема: програма може містити помилки через нерозуміння програмістом постановки задачі або специфікацій. В такому разі нерозуміння, ймовірно, поширюється і на тести. Це не означає, що програмісту заборонено тестувати свою програму. Проте процес буде ефективнішим і успішнішим, якщо ним займеться стороння людина.

Зауважимо, що ці аргументи не стосуються відладки (виправленню відомих помилок); її ефективніше проводити програмісту-автору, бо розбиратися в чужому коді значно складніше.

Потреба організації в незалежній групі, що тестує, може бути задоволена декількома способами. Група, що тестує, може бути виділена в повністю самостійний відділ організації. Альтернативою

цьому може стати група тестерів з групи забезпечення якості ПЗ (Software Quality Assurance Group) або група розробників, що спеціалізується на тестуванні, але в останньому випадку вона повинна бути максимально об'єктивною. Первинні обов'язки члена будь-якої з цих груп – тестування, а не розробка.

Нарешті, незалежність групи, що тестує, не має на увазі ворожих відносин з програмістами. Тестерам не варто грати в «квача» з розробниками. Вони повинні співпрацювати щоб забезпечити покупцю продукт найвищої якості.

Для того, щоб мати змогу перейти від ручного тестування до автоматичного була сформульована наступна пара принципів.

Принцип 8. Тести повинні передбачати багаторазове виконання.

В оригіналі:

Принцип 7: Краще уникати одноразових варіантів тестів, якщо сама програма не одноразового використання.

Представлений вище Принцип 2 закликав тестера розглядати свою роботу, як роботу ученого-експериментатора. Принцип 8 вимагає точного запису умов тесту, всіх важливих подій, що виникли під час тесту, і уважного аналізу результату. Ця інформація безцінна для розробників при поверненні коду на відладку, бо вона дає змогу відтворити умови тесту. Також вона корисна для повторних тестів після виправлення дефекту. Багаторазове виконання тесту відбувається і при регресивному тестуванні (поворотне тестування зміненої програми) нових версій ПЗ. Вчені знають, що їх експерименти повторюватимуть інші люди, це повинні знати і тестери.

Наступний принцип може здаватися очевидним, проте про нього часто забувають.

Принцип 9. Тестування слід планувати.

План треба розробляти для кожного етапу тестування, а мета повинна бути описана у відповідній його частині. Вона має бути задана настільки чітко, наскільки це можливо. Плани з описаною в них метою необхідні для адекватного розподілу часу і ресурсів на тести, а також для спостереження і управління процесом тестування. Планування тесту повинно бути внесено в життєвий цикл програми (Принцип 10) і скоординовано із плануванням проекту. Тестери не зможуть працювати над компонентом в заздалегідь обумовлений день, якщо розробники на той час його не доробили. Тому повинні бути обумовлені ризики тестування. Наприклад, імовірність затримки надання програмних компонентів, наявність компонентів складних для тесту, потреба в додатковому навчанні використанню нових інструментів. Шаблон плану тесту повинен бути доступний керівнику для грамотної розробки плану відповідно до політики і стандартів організації. Уважне планування тестів виключає некорисні одноразові тести та неефективні, незаплановані цикли «тест – виправлення – повторний тест», які часто затримують терміни виходу програми.

Принцип 10. Слід включати процес тестування в життєвий цикл ПЗ.

Якщо розпочинати тестування лише після того як програмний код вже повністю написано, то, з точки зору часових та фінансових витрат, воно не буде достатньо ефективним. Планування процесу тестування, як стверджує принцип 10, повинно бути включено в життєвий цикл програми, починаючи з етапу аналізу вимог і проводитися надалі, паралельно розробці. Додатково до планування, можна проводити різні тести на ранніх етапах, використовуючи прототипи, наприклад, перевірки на зручність використання. Ці процеси можуть тривати до самого випуску програми. Можливо

використання різних моделей впровадження тестування в життєвий цикл ПЗ.

Останній принцип є узагальнюючим і відображає підхід до роботи тестера

Принцип 11 (10). Тестування, це завдання, що вимагає творчості та розумової праці.

Труднощі роботи тестера, якщо їх деталізувати, полягають в наступному:

- *Тестеру необхідно мати вичерпні знання в області програмної інженерії.*
- *Тестеру необхідно мати досвід і системні знання у області специфіки, проектування і розробки ПЗ.*
- *Тестер повинен справлятися з багатьма дрібними задачами.*
- *Тестер повинен розбиратися у видах помилок і знати, де вони найчастіше можуть зустрічатися.*
- *Тестер повинен міркувати як вчений, висловлюючи припущення про наявність конкретних типів дефектів.*
- *Тестер повинен добре орієнтуватися в характерних для тестованого ПЗ дефектах. Це можливо при якісній освіті і досвіді тестування.*
- *Тестер повинен створювати і документувати контрольні приклади. Для розробки тестів необхідно вибирати вхідні дані з дуже широкого спектру. Вибрані дані повинні найбільшою імовірністю виявити дефект (Принцип 2). Необхідна обізнаність в цій області.*
- *Тестер повинен розробляти і зберігати процедури тесту для його виконання.*
- *Тестер повинен планувати оптимальну витрату ресурсів.*
- *Тестер повинен проводити всі випробування і протоколювати результати.*

- *Тестер повинен аналізувати результати і вирішувати, вважати тест успішним чи ні. Це вимагає розуміння та відстежування величезної кількості детальної інформації. Тестеру також потрібно збирати і аналізувати пов'язані з тестами вимірювання.*
- *Тестер повинен вчитися використовувати різні інструменти тестування, включаючи новітні розробки.*
- *Тестеру необхідно співпрацювати з інженерами, дизайнерами і розробникам. Також слід підтримувати контакт із клієнтами і користувачами.*
- *Тестер повинен підтримувати свої знання на високому рівні, регулярно підвищуючи свою кваліфікацію.*

ЛІТЕРАТУРА

1. Основы инженерии качества программных систем [Текст] : монография / Ф.И. Андон, Г.И. Коваль, Т.М. Коротун, В.Ю. Суслов ; НАН Украины. Ин-т прогр. систем. — К. : Академпериодика, 2002. — 502 с. — ISBN 966-8002-41-5
2. The Economic Impacts of Inadequate Infrastructure for Software Testing Research [Electronic resource]. Triangle Institute, NIST Planning Report № 02-3, RTI Health, Social, and Economics Research. — Research Triangle Park, NC, May 2002 — Mode of access: http://spinroot.com/spin/Doc/course/NISTreport02-3.pdf
3. Gates B. Trustworthy Computing [Electronic resource] / B.Gates. — Microsoft Corporation, Executive E-mail, July 18, 2002. — Mode of access: http://www.microsoft.com/mscorp/execmail/2002/07-18twc.mspx
4. Фокс, Дж. Программное обеспечение и его разработка [Текст] / Дж. Фокс. — Пер. с англ. — М.: Мир, 1985. — 368 с.
5. Майерс Г. Надежность программного обеспечения [Текст] / Г. Майерс. — Пер. с англ. под ред. В. Ш. Кауфмана. — М.: Мир, 1980. — 360 с.
6. Майерс Г. Искусство тестирования программ [Текст] / Г. Майерс. — Пер. — М.: Финансы и статистика, 1982. — 172 с.
7. Myers G.J. The Art Of Software Testing [Text] / G.J. Myers — New York: John Wiley & Sons, Inc., 2004. — 254 p. — ISBN 0-471-46912-2.
8. Молодцова О.П. Управління якістю програмної продукції [Текст]: навчальний посібник / О.П. Молодцова. — К. КНЕУ, 2001. — 248 с. — ISBN 966-574-230-2

9. Липаев В.В. Качество программного обеспечения [Текст] / В.В. Липаев. — М.: Финансы и статистика, 1983. — 263 с.
10. Липаев В.В. Тестирование программ [Текст] / В.В. Липаев. — М.: Радио и связь, 1986. — 296 с.
11. Дейкстра Дисциплина программирования [Текст] / Э. Дейкстра ; пер. с англ. И. Х. Зусман ; ред. Э. З. Любимский. — М. : Мир, 1978. — 275 с. — (Математическое обеспечение).
12. ДСТУ 2844-94. Програмні засоби ЕОМ. Забезпечення якості. Терміни та визначення [Текст]. — Введ. 1.08.1995. — К.: Держстандарт України, 1995. — 57 с.
13. ДСТУ 2850-94. Програмні засоби ЕОМ. Показники і методи оцінювання якості [Текст]. — К.: Держстандарт України, 1994. – 20 с.
14. ISO/IEC 9126. 2001. Software engineering – Software product quality – Part 1: Quality model. Part 2: External metrics. Part 3: Internal metric. Part 4: Quality in use metrics [Text] — Geneva, Switzerland: International Organization for Standardization.
15. Laprie J.C. Dependable Computing and Fault Tolerance: Concepts and Terminology [Text] / J.C. Laprie // Proc. 15th IEEE Int'l Symp. Fault-Tolerant Computing (FTCS-15) — Ann Arbor, MI, June 1985. — pp. 2-11.
16. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем [Текст] / Б. Бейзер. — СбП: Питер, 2004. — 320 с. — ISBN 5-94723-698-2
17. Patton R. Software Testing [Text] / R. Patton. — 2nd Edn. — Indianapolis: Sams, 2005. — 408p. — ISBN 0672327988